Domino: Trigger-based Programming Framework in Cloud

Dong Dai
daidong@mail.ustc.edu.cn

Li Xi
llxx@ustc.edu.cn

Lu Kun
local@mail.ustc.edu.cn

Mingming Sun
smile@mail.ustc.edu.cn

Chao Wang
saint@mail.ustc.edu.cn

Xuehai Zhou
xhzhou@ustc.edu.cn

University of Science and Technology of China
SuZhou Advanced Institution of USTC

ABSTRACT

Along with the rapid development of cloud computing, more and more applications are moving to a distributed fasion to solve problems brought by the 'Big Data'. These applications usually contain complex iterative or incremental procedures and have a more strict requirement on finish time, which make them hard to be designed and implemeted based on current cloud programming models. Although lots of models have been proposed recently, like Pregel, Spark, Storm, or GraphLab etc., we still believe the problem is still there especially when you need to consider the fault tolerance and recovery for your applications. So, in this paper, we propose a trigger-based programming model for cloud named Domino. It integrates with HBase storage system, which is popular in current cloud infrastucture as its high random read/write performance and high scalability, to provide developers the ability to write complex large scale applications especially the iterative and incremental ones. We show the simplity of trigger-based programming model based on describing our design and implementation of Domino. And by realizing some well-known algorithms including PageRank and distributed crawler using Domino model and having tests on their performance, we also show the general and high performance of Domino model. The Domino version applications can easily outperform MapReduce version tenfold and with the great benefits on fault tolerance and realtime recovery ability which was brought by the trigger-based model. Through our research, we believe the trigger-based programming model is a stirring solution for large scale data processing in cloud enviornment.

1. INTRODUCTION

The availability of Cloud Computing services like Amazon EC2 and Windows Azure provide an on-demand access to affordable large-scale computing resources without substantial upfront investments. However, designing and implementing different kinds of scalable applications to fully utilize the cloud can be prohibitively challenging requiring domain experts to address race conditions, deadlocks, distributed state while simultaneously concentrating on the problem itself. To help shield application programmers from the complexity of distribution, lots of programming frameworks have been developed recently.

Some existing frameworks are synchronous data-flow programming models. These models target synchronous computation where the data flow was read and processed with global synchronization between different phases or iterations. For example, MapReduce[11], Dryad[16] and some variations based on them[6, 12, 34, 35, 33]. These frameworks are ideally suited for bulk-processing without visiting any global shared state. Applications like *wordcount*, *sorting* can be easily implemented in such model but incremental or iterative applications are not suitable. Another kind of model can be described as data-centric programming model, like Piccolo[28] and Pregel[22]. They rely on global barriers to execute applications round by round. But, in such models, the asynchronous solutions which are more efficient on many problems can not be implemented efficiently. There are also asynchronous models which do not need explicit synchronization when applications execute like S4[26], Ooloong[24] etc, but most of them are not general enough as a standalone framework. Recently, GraphLab[19] is popluar. It is also an asynchronous framework, but is designed for graph computation and used in many machine learning and data mining applications (alternative least squares, loopy belief propagation and EM etc.). It is not so straitforward to transfer any application into a graph computing problem and this limits GraphLab's usage in many fields.

Incremental processing, in which applications only need to process many small updates each time to provide continuous correct results, have proven a convenient and efficient mechanism for many applications, like PageRank, distributed crawlers, and many online MLDM (machine learning and data mining) algorithms. However, the programming frameworks mentioned before can not handle these applications well. Percolator[27] and Oolong[24] provide an event-driven programming model to support this incremental computation. However, both of them are not designed and implemented as a general programming model and fail to provide whole solutions for challenges introduced by the large scale trigger-based systems.

This paper presents Domino, a trigger-based incremental

programming framework in cloud for writing general distributed high performance iterative or incremental applications across commodity machines. In Domino, programmers organize the computation into a series of triggers, which contain user-specified code blocks that can be invoked when the associate events were triggered. Both synchronous or asynchronous applications can be implemented easily under this programming model. Besides, during the whole runtime of the system, Domino uses *multi-version table* to keeps trace all the data processing status and trigger status so we can detect the failures and recover them in real time while application executing.

The contribution in this paper includes:

- We provide a genetic trigger-based programming framework which supports both synchronous or asynchronous model for iterative and incremental applications.
- We provide a distributed runtime system, which has great fault tolerance and realtime recovery ability based on multi-version tables

2. PROGRAMMING ABSTRACTION

Trigger-based framework follows the ECA model[23] (Event, Condition, Action model), which has been widely researched in active database field[7, 13, 17, 29] and implemented in many productions like Postgres[32], HiPAC[10], Sybase[9], Vbase[4] and OOPS[30].

ECA model means when certain events occur (*on event*), and some conditions are fulfilled (*if condition*), then some actions are executed automatically (*then action*). The event represents updates on the input dataset of applications. The condition controls the execution of users' programs. And the action contains the user-specified codes that would run concurrently across whole cluster. To demonstrate Domino abstraction more clearly, we will use the PageRank algorithm as a running example.

> PageRank algorithm calculates the priority of web pages. In PageRank, we use a transition matrix M to represent the whole web graph. To get the rank value of every page we need to calculate the rank vector v use this equation:
>
> $$v^{new} = \beta M v^{old} + (1 - \beta)e/n \qquad (1)$$
>
> Where β is a chosen constant, usually in the range 0.8 to 0.9, e is a vector of all 1s with the appreciate number of components, and n is the number of nodes in the web graph. PageRank runs iteratively until the difference between v_{new} and v_{old} less than a small value ϵ.

2.1 Event

Update event, as the main event source in Domino, represents the modification on the under layer storage entity, in our case, is HBase[8]. In HBase, data entity was stored in tables. Each table contains billions of rows and thousands of column families with millions of columns. The row key can be arbitrary strings, and all reads or writes inside a single row key is atomic. Column are grouped into column families, which form the basic unit of HBase. Access control and both disk and memory accounting are performed at the column-family level. All data stored in a column family is usually of the same type. A column family must be created before data can be stored under any column in the family. After the family was created, any column key within the family can be used, and the column is named using famliy:qualifier syntax. A row and a column define a cell, each cell in a HBase can contain multiple version data, and these versions are indexed by timestamp which are 64-bit integers. These timestamps can be assigned by HBase, in which case they represent 'real time' in microseconds, or be explicitly assigned by client applications.

In Domino, Programmers can issue triggers to monitor any individual table, a specific column family or a column. Then any update on these entities would generate an *update* event which could fire next phase execution. Events generated by Domino can be divided into simple and complex events. Complex events are mostly based on simple ones, and simple events can be divided as follows: 1) Time events including a)absolute - certain point of time; b) periodic - every day, month, etc.; c) relative - 30 minutes after something else has happened. 2) Update events: updates on data piece; 3) abstract events: some user defined events like create table or delete table. Complex events are assemble of different simple events. However, in Domino, the programming model mainly cares about the update events because most the applications we want to program are trying to react towards the modification on the dataset. Time events are indeed neccseary in some periodical appliations, but we can easily to setup 'cron' jobs for them insead of seting up triggers, so, we will mainly describe the *update* event in this paper.

To process the exited data before triggers are submitted to Domino, an optimization was proposed: when an *update* trigger was set on a table, all the existing data will fire $0 \rightarrow 1$ events. This solves the cold start problem of the trigger model.

> As Table.1 shown, PageRank algorithm only needs to setup a *update* event on the PageRank column of the *WebPages* table. Any change on the rank value of page will case a recalculation of all its relevant pages.

Table 1: *WebPages table* in HBase

WebPage	*PageRank Value*	*Out Edges*	...
p_1	0.5(default)	$p_{11}, p_{12}, p_{13}, ...$	
p_2	0.5(default)	$p_{112}, p_{21}, p_{32}, ...$	
...	...	... , ...	

2.2 Condition

Condition is a user-specified code fired on specific events. It returns *true* or *false* to denote whether users conditions were fulfilled. It takes at least three critical responsibilities in Domino. Firstly, the triggers would be persistent in Domino until explicitly cancelled, but triggers should be terminated before cancelled by hand when final results have been calculated. For example, in the PageRank algorithm, triggers should stop when result converges. In such cases, stop condition can help programmers terminate applications. Besides, *update* Event was generated by the modifications on HBase in Domino. If these modifications are too frequent, Domino's execution engine may consume all resources of a cluster. To avoid this situation, each trigger

by default has a *interval* condition to control the execution frequency, also programmers can define their own *interval* condition too. At last, *select* condition, which means select part of the fired events to process, is also important for programmers to write flexible and high efficient applications.

> In PageRank, for a web page wp_i, when the difference between $rank'_{wp_i}$ and $rank_{wp_i}$ is less than ϵ, we should stop to alter its neighbors. So the stop condition is: **cond:true** $[r_{new} - r_{old} \leq \epsilon]$

2.3 Action

Action is the core of a trigger-based model, it is a code block written by programmers. Rather than using message passing or data flow model, Domino adopts the data-centric model which allows the user-defined actions completely freedom to read modify any of the data stored in HBase especially the relevant data which triggered this action. This simplifies user code and eliminated the need for the users to reason about the movement of data flow. Besides, by providing the *Gathered I/O* to delay the writing to HBase guarantee the atomic attributes of writes inside one action, Domino supports a high performance and easy used distributed writes ability. We will describe the design and implementation of the I/O parts in next section.

> Action for modification of pagerank value of a page is quite simple: for any page, change the pagerank weights for all its neighbors by adding the page rank weight introduced by current page.

Algorithm 1 PageRank action function

Require: Event object of current row (e)
  $n \leftarrow e.outedges$ ▷ get edges number
  $w \leftarrow e.rank/n$ ▷ calculate each edge weight
  for page ∈ e.outedge[] **do** ▷ write into ***tips.1***
    lazy-write(pr-acc, page, curr-page, w)
  end for
  flush all written in lazy-write

There is an important difference between the existing event-driven programming models like Percolator or Oolong and Domino: how aggregation was performed. Taking PageRank as an example, the rank value of a specify page was generated by summing (aggregating) all its link-in edges' pagerank weights. In Oolong, programmers can provide a explicit, per-defined *accumulator function* to aggregate particle results, but it is too simple and limited to express complex computations. On the other hand, Google's Percolator does not provide any explicit aggregation method, so programmers need to find their own way to implement it, which is error-prone in distributed environment.

In Domino, we provide a design pattern named *accumulator pattern* for aggregating partial results. When programmers need aggregation, they can simply follow the *accumulator pattern*: firstly, submit an *accumulator* trigger alone with the original trigger; secondly, write all the values you need to aggregate from the original trigger into that *accumulator* trigger instead of HBase tables, just as ***tips.1*** in Alg.1 has shown; thirdly, write the action function in the *accumulator* trigger to finish any aggregation you want, like *sum*, *max*, etc. Inside the Domino, *accumulator pattern* works like this: the intermediate results generated in the original action functions will be written into a distributed table (t_{acc}), which is created automatically by Domino. Then the *accumulator* trigger that programmers proposed will monitor all the modifications in t_{acc} and process them according to your accumulator triggers.

> In PageRank application, the original action has written edge weights into the *pr-acc accumulator* trigger which finishs the aggregation works. The accumulator trigger action only needs to simply sum up all the pagerank weights generated from different link-in pages and write the result back into Table.1, and call for next phase execution.

Algorithm 2 PageRank accumulator action function *pr-acc*

Require: In-memory distributed table t_{acc}. page-id as the row-key.
Require: Each column represents the weights from one link-in edge.
  for $weight_i \in t_{acc}.columns$ **do**
    pr += $weight_i$
  end for
  flush pr back to *WebPages* table

Beside automatically creates a distrbuted table t_{acc}, the *accumulator* pattern in Domino also calls for a mechanism to handle the situation where different trigger instances from different servers asynchronously arrive and try to update the same location. To solve this problem, we introduce a serial of flexible synchronous models in Domino.

3. SYNCHRONIZATION MODEL

As we know, synchronous systems update all parameters simultaneously using parameter values from the previous phase as input. Asynchronous systems update parameters using the most recent parameter values. Although synchronous computation usually incurs costly performance penalties because their execution time is determined by the slowest machine, it is still necessary to make application correct.

To balance the synchronization needs and the performance of applications, Besides providing both synchronous and asynchronous models (Domino's nature model), Domino provides an **eventually synchronous** model to leverage performance of applications. Developers in Domino can choose any of these three models according to their needs.

3.1 Eventually Synchronous

In Domino, the eventually synchronous model is implemented using multi-version tables. As we have described, developers use the *accumulator triggers* to aggregate results from different actions of the original trigger. So, synchronization is represented as an *accumulator triggers* monitoring on a distributed HBase table: t_{acc}. The table is automatically created and named by Domino followed by the unique trigger instance id, and only be visible inside current trigger. As each cell in t_{acc} stored multiple versions data, we have the ability to make different executions eventuall synchronized by their version numbers.

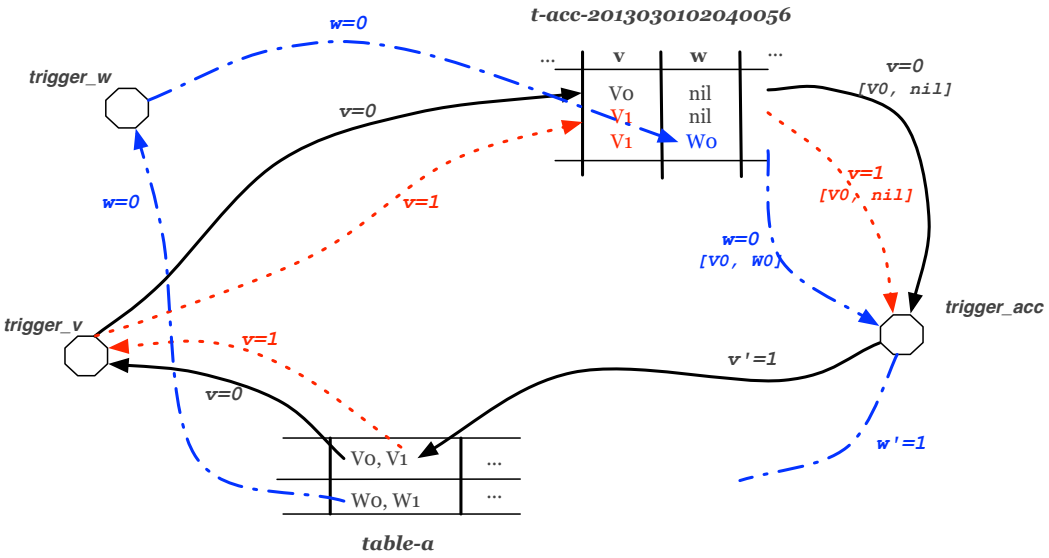

Figure 1: Multiple Version Execution Flow in Domino

In Domino, the version data, also can be called as *round id*, inherits from the *round id* of the trigger action which try to write the new value into t_{acc}. Each time, when an update with version v_i happened on t_{acc}, it will cause the action function of *accumulator triggers* executed. Inside this action function, we will read data generated from other actions as we need to synchronize them. To keep final result correct, Domino will only read the data which has the version number less or equal to v_i, generated results with version v_{i+1}, and write into HBase tables.

Fig.1 shown an execution procedure of a synchronous application. There are two trigger actions (*trigger_w, trigger_v*) fired by the *update* event on *table-a*, and an *accumulator* trigger (trigger_acc) to help guarantee their eventually synchronous. Distributed table *t-acc-2013030102040056* was created automatically by Domino. It shows how to synchronize the execution of *trigger_v* and a slower *trigger_w*: until *trigger_v* has execute its *1 round*, *trigger_w* began to run its *0 round*. So, each time an update is issued by *trigger_v*, the *accumulator* trigger will generate an intermediate result. During the execution, there are two intermediate results calculated using $[v_0, \text{nil}]$, $[v_1, \text{nil}]$. These results may be not correct, but it serves as the intermediate results which can be used when the partial results are meaningful. Eventually *trigger_w* will finish and write w_0, and the synchronized result with version number 1 which is calculated using $[v_0, w_0]$ will be written into *table-a*.

3.2 Strict Synchronous

The design and implementation of strict synchronization is simple in Domino. As we know, the strict synchronization means the next phase computation will not start until all previous phase executions have been finished. In Domino, as we have talked above, all the synchronization should be programmed following the *accumulator patter*: suppose we have different action instances from one trigger (say t_a) need to be synchronized, we shall setup an *accumulator* trigger(t_{acc}), and write code inside action function of t_{acc}. However, The main challenge is that the developers do not know how many action function instances from t_a are executed in the previous phase, so they do not know when should start t_{acc} for a synchronization. To solve this problem, Domino provides two APIs: **register(trigger-id)** and **wait-syn(trigger-id)** which can be used in action functions. In action function of t_a, developers register current round and trigger to Domino runtime. Current round id is generated by adding 1 to the round id of the event that fires current execution. In *accumulator* trigger t_{acc}, developers call **wait-sync** to wait for actions that have been registered to finish. This means actions in t_{acc} would not start until all actions of last round from trigger t_a finished.

4. DESIGN & IMPLEMENTATION

In this section we present the trigger-based computation framework design issues and challenges, and discuss the techniques required to achieve our goal. Domino was designed and implemented in Java based on HBase and worked as a plugin of HBase. However, it is not limited to HBase. Domino works well for other distributed storage system which can provide persistency and multiple versioned data. We are working on other version of Domino as future work now.

In Fig.2 and Fig.3, we provide a high level overview of a Domino system. When Domino is launched on a cluster, the instance of the Domino program starts on each machine accompany with HBase instance. There is one *TriggerMaster* node running the extra management part with the *HMaster* node in HBase like fig.2 shows, and all the other *TriggerWorker* nodes will run exactly on all other nodes with *HRegionServer* in HBase. The *TriggerWorkers* communicate with *TriggerMaster* using a synchronous RPC protocol over TCP/IP just like HBase does. The first process has registered itself as the master has the additional responsibility of being the *TriggerMaster* machine. Due to the symmetric design of the distributed runtime, there is no centralized bottleneck.

At launch, the *Manager* component of *TriggerMaster* will send a request to HBase[1] to ask for the location information of relevant tables through the *Sync RPC Connection*

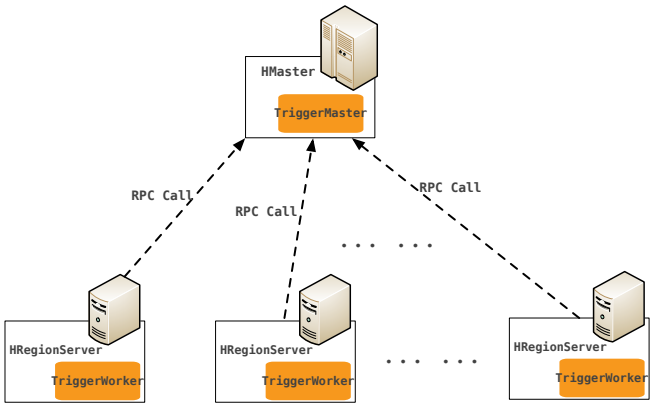

Figure 2: Architecture of the Domino cluster running on a HBase cluster.

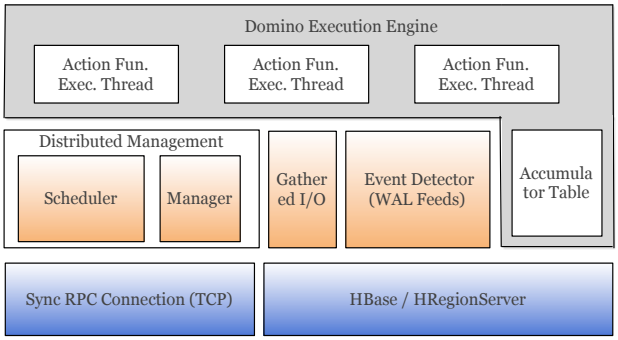

Figure 3: A block diagram of the parts of the Domino instance. Each block in the diagram makes use of the blocks below it.

component. Then the trigger instance will be assigned to different servers, and each trigger instance is responsible for a partition of the table that is monitored by a trigger.

Besides, as fig.3 shows, each process contains a *Event Detector* to detect the modifications on HBase table. In Domino, we use **WAL** (write-ahead-log) **Feed** to accomplish this work. We will describe it in detail in section 4.1. The *TriggerMaster* also contains a scheduler that manages the trigger assignment: when developers submitted a trigger, the scheduler assign the trigger to the servers whose loads are more reasonable in future, at the time this trigger begin to execute. During runtime, the scheduler can reassign the running triggers according to current machine status also to balance the whole cluster.

4.1 WAL Feeds Event Detecting

Domino *Manager* component automatically assigned triggers to servers that the monitored dataset was stored in. So the local modification detector for *Update* event is sufficient.

In HBase, the data was firstly written into *Memstore* instead of disk to accelerate the response time. Due to the *Memstore* resided in RAM, it introduces an element of risk. To help mitigate this risk, HBase saves updates in a *WAL* before writing the information to memstore. Currently, the *WAL* of HBase contains information includes the *timestamp, key, value*. To work with our Domino, we modify the *WAL* relevant code and add more information: *old key/value pairs,*

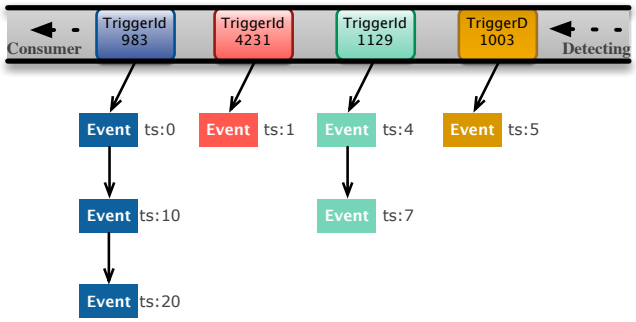

Figure 4: Domino Event Queue. If two events belong to the same trigger, they would be placed at the same position of the event queue. The priority is based on the timestamp. Consumers get all the events belong to one trigger each time to reduce the latency caused by context switch.

last access timestamp etc.

On each *TriggerWorker*, Domino's *Event Detector* registers itself as *WALActionListener* on the write-ahead-log files to detect the append actions on it. Using the *epoll* lib[18], the processing code will be executed once the WAL files were updated. Once the local *Event Detector* noticed modifications, it built an *Event* object containing the information collected from the *WAL* logs, like the old/new key/value pairs, the old/new timestamps, and other environment variables. This *Event* object will be sent to an event queue just like Fig.4 have shown. The queue is attached with a consumer waiting for processing the new event.

At end, the triggered events will be processed by the user-specified action functions. There are some pre-allocated threads in each machine waiting for executing these action functions. In Domino, we extend the Java concurrent thread lib to a managed thread pool, which can trace each thread's status, restart failed threads. The action function is flexible to access any data in HBase which introduces lots of consistent and performance issues. Domino provides *Gathered I/O* to solve these problems.

4.2 Gathered I/O

In Domino, user-defined actions can access any data following the data-centric model. So, when different triggers are trying to write into the same location simultaneously, there would be a consistent problem. Besides, frequently call the time-consuming data access API will dramatically deduce the performance of applications. To address these problems, we provide a *WritePrepare* class to encapsulate all writes and delay the real writing till the action returns. Just like the MapReduce model, most the writes are issued at the end of reduce functions, the action in Domino submitted the writes at the end of action functions.

In action functions, all users' writes should be written into *WritePrepared*. Before exiting action functions, all data cached in *WritePrepared* will be flushed into HBase table according to their initial order of calling *append* method. According our lazy-write strategy, we should know that all the write operations in Domino's action function are only visible inside this action until the flush method is finished.

Each time we call *flush* method of *WritePrepared*, it will register itself to a global consistency service (ZooKeeper)

and get a global sequential number S_f. If there are two writing requests from different triggers of Domino and they are not confilict, then HBase will not sense any difference from the ordinary writes. However, if they are conflict with different sequential number $S_{f_1} < S_{f_2}$, then the second flush would be suspended until the first one has finished, and during the suspending, all the succesful writes by S_{f_2} will be over-writted. This strategy does not mean all writes are sequential, but just means Domino chooses the write operations with the currently minimal sequential number, and once it started, it will not be preempted by other writes.

Using the *WritePrepared*, we leverage the performance of write performance of Domino application and we can guarantee that all writes from different action functions would not intersect each other. And we also guarantee that all the writes would not be overwritten by other write requests from earlier round.

4.3 Fault Tolerance & Failure Recovery

Fault tolerance is a critical challenge for large scale platforms like many streaming processing models or other realtime computation models. However, in Domino, we are able to provide a remarkable fault tolerance and nearly realtime failure recovery ability based on the trigger-based model, by which all the intermediate results should be stored in persistent storage HBase instead of only existing in memory.

First of all, Domino uses HBase as the basic fault tolerance infrastructure. HBase is a fault-tolerant distributed storage system, which help Domino store all triggers that users submitted, all data that has been written by Domino, and all the status of existed triggers. As we know, HBase cluster can sustain half *ZooKeeper* servers fail, so the Domino also runs well in such condition.

Except for these static fault tolerance by storing status information, we want the running status can also be fault tolerant and recoverable.

In the situation the whole physical node crashed while triggers are executing, the HBase instance on that server will also crashed. The heartbeat signal will tell $TriggerMaster$ that failure happens, then all writes in HBase will be directed to another backup node selected by HBase, and HBase will re-construct the data into memory in the backup node. Domino will also sense this failure and move the triggers on the failed node to the backup node immediately. This means that all the writes after node failure will be processed by the migrated triggers. For those triggers whose actions are running while failure happens, as we use *Gathered I/O*, all the outputs were cached in *WritePrepared* instance, which in Domino is a znode in *ZooKeeper* with a global id (G_{id}). If *flush* has been called when failure happens, we will know that the *WritePrepared* has started to flush itself into HBase, then we will continue this flushing in the backup node according to the information recored in *ZooKeeper*. If *flush* has not been called, we will know there has been any modification on HBase yet, we can safely re-run the action function again, we can simply run these failed actions on the backup node using data with correct version number.

In pervious section, when we introduce the *Gathered I/O*, we do not describe the *WritePrepared* instance in detail. In fact, each *WritePrepared* instance will create a znode in *ZooKeeper* namespace with a global unique id G_{id}, and also a local ordered queue in memory. Whenever developers call *append* method, Domino will enqueue the new appened write into the ordered queue in memory and update value of znode with key G_{id}. Only if these two actions are finished, the *append* method returns succesfully. When *flush* method is called, Domino will flush the elements in the ordered queue into HBase one by one. Each time we successfully flush a write into HBase, we will record the sequence (seq_i) of it into *ZooKeeper*. This seq_i will be written into the same znode of the *WritePrepared* instance obtains. Using this strategy, we only need to transfer the heavy write operations once(from *WritePrepared* to *ZooKeeper*) and keep all status is recorded persistently: whenever failure happens, we can continously flush data by current sucessful sequence id and all the writes in *WritePrepared* instance.

5. EVALUATION

We evaluated Domino on perviously mentioned state-of-the-art distributed applications: distributed crawler, PageRank, collaborative filtering for recommendation systems and K-means parallel computing. Equivalent MapReduce implementations were also evaluated on these applications. Unfortunately, we can not compare our Domino with the successful Percolator since it is not public available and current open source implementations do not scale even the smaller problems we considered. However, we can still get the basic idea that the performance advantages comparing with Percolator should be obvious because Percolator uses modified Bigtable to provide distributed transaction to guarantee consistency, which will easily slow the execution of tasks especially when trigger number is continuously increasing. Other trigger-based systems like Oolong is not included in these experiments too because the advantages of Domino comparing with them are not performance. The advantage of Domino mainly are that we support synchronize ability for event-driven systems and provides realtime fault-tolerance ability.

According to our experiments, we have some principle findings here:

- Based on the experiments on asynchronous and synchronous applications, Domino outperforms Hadoop easily at least by 10X.
- The performance of applications implemented in Domino scales well along with the machine numbers increasing. This indicates Domino provide good scalability.
- The Domino abstraction can easily express the PageRank and lots of machine learning algorithms.

5.1 Test Setup

Most experiments were performed using our local cluster of 9 machines: all of the machines have Xeon dual-core 2.53 GHz processor with 6GB memory. All machines are connected via a commodity gigabit ethernet switch. For scaling experiments, we vary the input size of different applications. The PageRank's default input data set contains 1 million person's pages which is generated by our distributed crawler, but in the scaling experiments, we generate other persons randomly up to 1 billion persons.

5.2 Performance Degration of HBase

As we have described in previous sections, the Domino system is implemented based on HBase by modifying its WAL append operations, adding periodically scan threads

on HBase tables. So the performance degration mainly introduced by these modificatoin on HBase. Another main source of performance degration comes from users' trigger action functions execution inside the JVM of each *HRegionServer* in HBase. In this experiment, we will compare this performance between HBase and Domino on our 9 nodes local cluster.

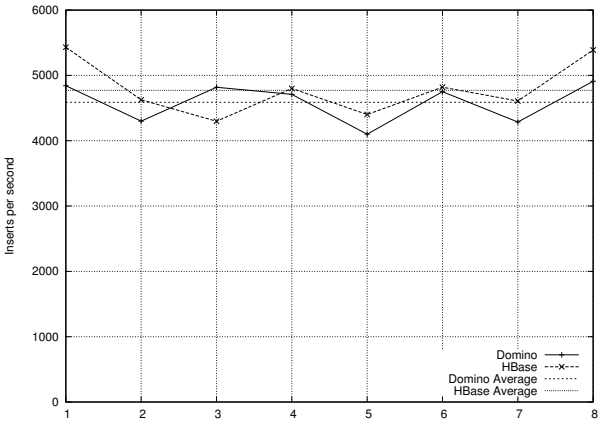

Figure 5: Performance Comparision between Domino and HBase while no trigger is running

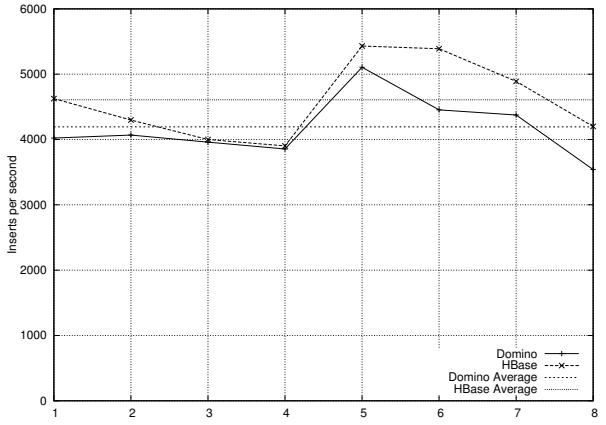

Figure 6: Performance Comparision between Domino and HBase while triggers are heavily fired

We use the **PerformanceEvaluation**[15] which was introduced in HBase-399 to evaluate the performance of HBase. The whole test program was running as a MapReduce job. In this program, given a parameter n clients, it will start $10*n$ map tasks, each client inserts 1 millions rows (each row contains 1000 bytes). Accordingly, each map task will insert 10,000 rows into our HBase table. As the reduce phase only sums all the row lines number written by different maps, in the final result, we dismiss reduce time. To compare Domino and HBase performance, we average the execution time of all the map tasks and calculate the average write speed per map task.

Fig.5 shows the comparision results (**8 different exp. results**) while there is not any trigger running in Domino: all the performance penaty comes from our *WAL Feeds* implementation. We can easily find out that there indeed is a small performance difference (4%) between HBase and Domino. And, Fig.6 shows the comparision results (**8 different exp. results**) while triggers are running in Domino. The trigger was setup on the table which *PerformanceEvaluation* inserts on. Whenever there is an update on that table, trigger will execute action to do some simple operations to consume a small CPU resources. From fig.6, we can find out even under a events heavily fired situation, the performance difference between HBase and Domino is less than 10%. According to these results, we can conclude that the performance degration introduced by Domino and its triggers is small enough to make Domino a high efficient framework.

5.3 Scaling Performance

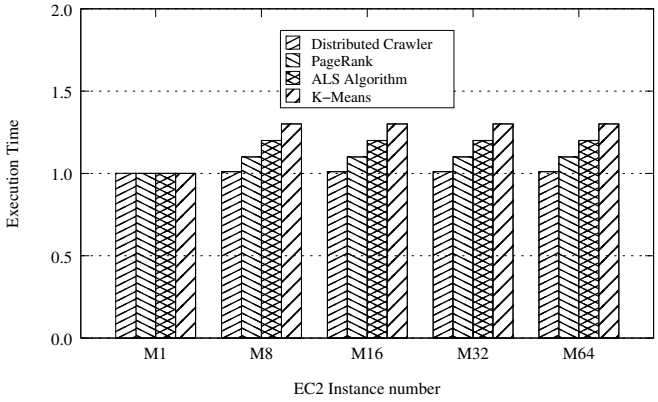

Figure 7: Applications performance while input data size increase

Fig.7 shows application performance as the number of servers increases from 1 to 64 for the default input size in 64 nodes, add the increasing input size is also increasing to keep the amount of computation per server fixed with increasing server numbers. We scale the input size linearly. The ideal result would be a constant running time when input size increases. As Fig.7 shows, the achieved scaling for all applications is within 30% of the ideal number.

5.4 Performance Comparing with MapReduce

We implemented PageRank in Hadoop to compare their performance against that of Domino. All the experiments are finished on our 9 nodes local cluster with the default input dataset. In our experiment, we run all the Domino version applications (using in-memory speedup) and Hadoop version on 1 nodes, 3 nodes, and 9 nodes to check the performance scalability of Domino. From fig.8 we can easily see that Domino's PageRank achieves about 10x speedup over MapReduce solution (each has 5 iterations) while re-quiesce a 1M vertex graph.

5.5 Incremental Performance

Our Domino programming framework provides the supports for incremental applications. The PageRank benchmark provides a good basis for testing the effect of incremental programming supports because the web graphs will be continuously changing both in our experiments and in real world.

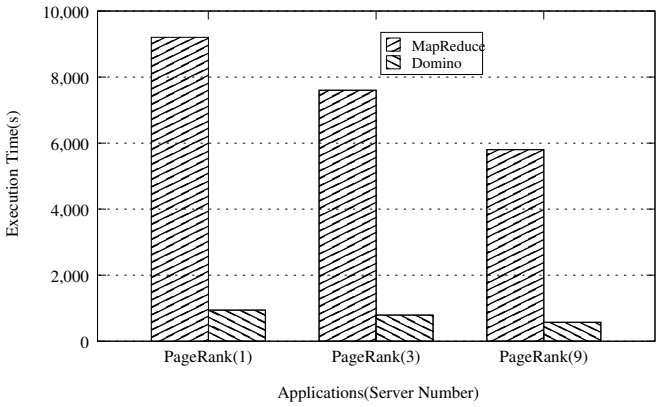

Figure 8: Performance Comparison of PageRank between Domino and MapReduce implementation in 1/3/9 nodes local cluster.

Fig.9 shows the comparison between Domino and MapReduce solution to calculate the PageRank value while pages are partially changed, the y-axis means the ratio of partial computing and whole computing time. In the basic MapReduce semantic, whenever the amount of additional pages exceed a threshold or we reach the time interval, the MapReduce application will run a new iteration over all the dataset again. However, in Domino, application only needs to calculate the updated pages and some relevant pages whose PageRank value was affected by these changed pages. So, Domino's incremental computation time scales with the size of the changes: if only 100 pages were changed, our Domino application can give the result instantly.

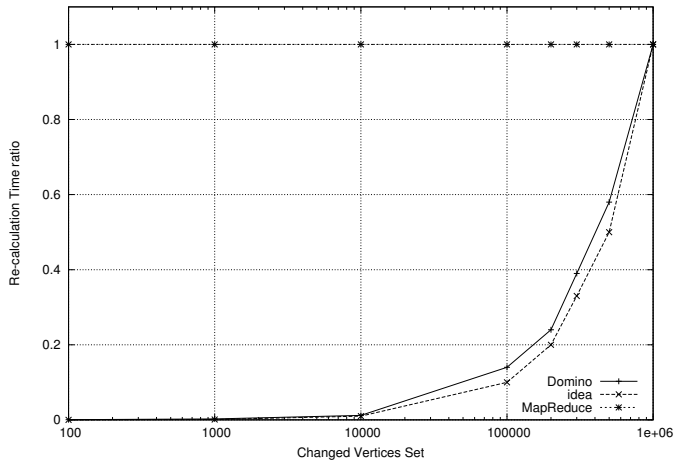

Figure 9: Incremental PageRank Comparing with MapReduce

6. CONCLUSION & FUTURE WORKS

With the *eventually synchronous* mechanism, Domino simplify the applications logic for both asynchronous and synchronous applications. To leverage the performance of our trigger-based programming model, Domino also provides gathered I/O mechanism to help process large scale of concurrent writes. Facing the frequent failures in cloud environment, Domino propose strong failure recovery speed based on multi-version storage systems. According to our experiments and use cases in different fields, we noticed that the extended trigger-based programming model is much more general than triggers in databases, and have more advantages in many specific applications.

Although Domino has shown the potential of general trigger-based programming model in cloud environment, there are still lots of challenges remained. The scheduler in such system is a complex and necessary component. Due to the difference from submit-and-run jobs, Domino triggers are not instantly running after their submission, that is to say, they need specific events to fire them. So, the scheduler needs to be aware of these future events and schedule triggers into different nodes to balance their loads. Another important problem we will work on is leverage the performance of current Domino implementation. Besides, Domino is a programming framework which should be able to work with different distributed storage systems whenever this storage system fits the needs of Domino. We are trying to combine Domino with different storage systems.

7. REFERENCES

[1] Hbase project. In *http://hbase.apache.org*.
[2] Mahout project. In *http://mahout.apache.org*.
[3] Storm project. In *http://storm-project.net/*.
[4] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *ACM Sigplan Notices*, volume 22, pages 430–440. ACM, 1987.
[5] D.P. Bertsekas and J.N. Tsitsiklis. Parallel and distributed computation. 1989.
[6] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.
[7] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the international conference on very large data bases*, pages 606–606. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994.
[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
[9] M. Darnovsky and J. Bowman. Transact-sql user's guide. *Sybase Inc., Doc*, pages 3231–2, 1987.
[10] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, et al. The hipac project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1):51–70, 1988.
[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[13] N. Gehani and HV Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB)*, pages 327–336, 1991.

[14] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. Aistats, 2009.

[15] HBasePerformanceScript. https://issues.apache.org/jira/browse/hbase-399.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[17] U. Jaeger and J. Obermaier. Parallel event detection in active database systems: The heart of the matter. *Active, Real-Time, and Temporal Database Systems*, pages 159–175, 1999.

[18] D. Libenzi. Linux epoll patch, 2006.

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[21] W.G. Macready, A.G. Siapas, and S.A. Kauffman. Criticality and parallelism in combinatorial optimization. *SCIENCE-NEW YORK THEN WASHINGTON-*, pages 56–58, 1996.

[22] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.

[23] D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM Sigmod Record*, 18(2):215–224, 1989.

[24] C. Mitchell, R. Power, and J. Li. Oolong: asynchronous distributed applications made easy. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 11. ACM, 2012.

[25] R.M. Neal and G.E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. *NATO ASI SERIES D BEHAVIOURAL AND SOCIAL SCIENCES*, 89:355–370, 1998.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[27] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–15. USENIX Association, 2010.

[28] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2010.

[29] K. Rabuzin, M. Maleković, and A. Lovrenčić. The theory of active databases vs. the sql standard. In *The Proceedings of 18th International Conference on Information and Intelligent Systems*, pages 49–54, 2007.

[30] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl, and R. Meyer. Oops-an object oriented programming system with integrated data management facility. In *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 118–125. IEEE, 1988.

[31] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[32] M. Stonebraker, E.N. Hanson, and S. Potamianos. The postgres rule manager. *Software Engineering, IEEE Transactions on*, 14(7):897–907, 1988.

[33] H. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[34] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10. USENIX Association, 2010.

[35] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 13. ACM, 2011.

[36] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. *Algorithmic Aspects in Information and Management*, pages 337–348, 2008.